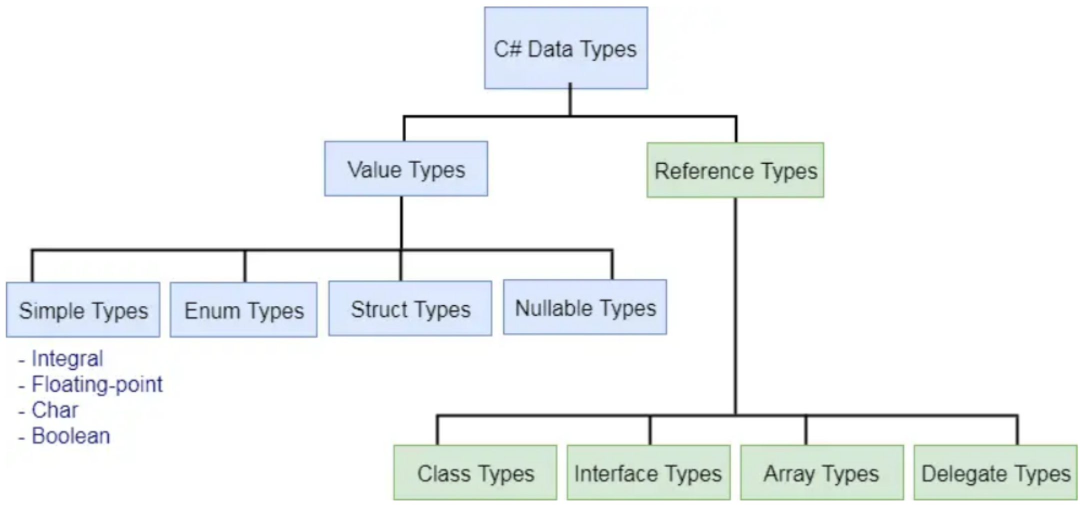
C# - Data Types

- C# is a strongly typed language.
- It means we must declare the type of a variable that indicates the kind of values it is going to store, such as integer, float, decimal, text, etc.

Variables of Different Data Types

- string stringVar = "Hello World!!";
- int intVar = 100;
- float floatVar = 10.2f;
- char charVar = 'A';
- bool boolVar = true;



Predefined Data Types in C#

C# includes some predefined value types and reference types. The following table lists predefined data types:

Type	Description	Range	Suffix
byte	8-bit unsigned integer	0 to 255	
sbyte	8-bit signed integer	-128 to 127	
short	16-bit signed integer	-32,768 to 32,767	
ushort	16-bit unsigned integer	0 to 65,535	
int	32-bit signed integer	-2,147,483,648 to 2,147,483,647	
uint	32-bit unsigned integer	0 to 4,294,967,295	u

Type	Description	Range	Suffix
long	64-bit signed integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	l
ulong	64-bit unsigned integer	0 to 18,446,744,073,709,551,615	ul
float	32-bit Single-precision floating point type	-3.402823e38 to 3.402823e38	f
double	64-bit double-precision floating point type	-1.79769313486232e308 to 1.79769313486232e308	d
decimal	128-bit decimal type for financial and monetary calculations	(+ or -)1.0 x 10e-28 to 7.9 x 10e28	m
char	16-bit single Unicode character	Any valid character, e.g. a,*, \x0058 (hex), or\u0058 (Unicode)	
bool	8-bit logical true/false value	True or False	
object	Base type of all other types.		
string	A sequence of Unicode characters		
DateTime	Represents date and time	0:00:00am 1/1/01 to 11:59:59pm 12/31/9999	

Example: Compile time error

```
// compile time error: Cannot implicitly convert type 'long' to 'int'.
int i = 21474836470;
```

The value of unsigned integers, long, float, double, and decimal type must be suffix by u,l,f,d, and m, respectively.

Example: Value Suffix

```
uint ui = 100u;
float fl = 10.2f;
long l = 45755452222222l;
ulong ul = 45755452222222ul;
double d = 11452222.555d;
decimal mon = 1000.15m;
```

C# Pass Value Type by Value

In c#, if we pass a value type variable from one method to another method, the system will create a separate copy for the variable in another method.

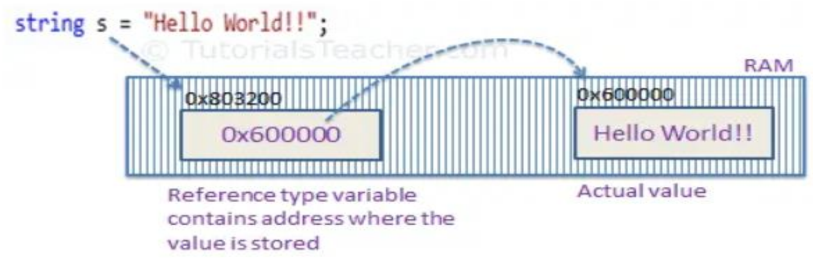
If we make changes to the variable in one method, it won’t affect the variable in another method.

```
using System;

namespace Tutor
{
   class Program
   {
        static void Square(int a, int b)
        {
           a = a * a;
           b = b * b;
           Console.WriteLine(a + " " + b);
        }
        static void Main(string[] args)
        {
        int num1 = 5;
        int num2 = 10;
        Console.WriteLine(num1 + " " + num2);
        Square(num1, num2);
        Console.WriteLine(num1 + " " + num2);
        Console.WriteLine("Press Enter Key to Exit..");
        Console.ReadLine();
        }
   }
}
```

C# Reference types

- In c#, **Reference Types** will contain a pointer that points to another memory location that holds the data.
- The **Reference Types** won’t store the variable value directly in its memory.
- Instead, it will store the memory address of the variable value to indicate where the value is being stored.



string s = "Hello World!!";
© TutorialsTeacher.com
RAM
0x803200
0x600000
0x600000
Hello World!!
Reference type variable contains address where the value is stored
Actual value

Memory Allocation of Reference Type Variable

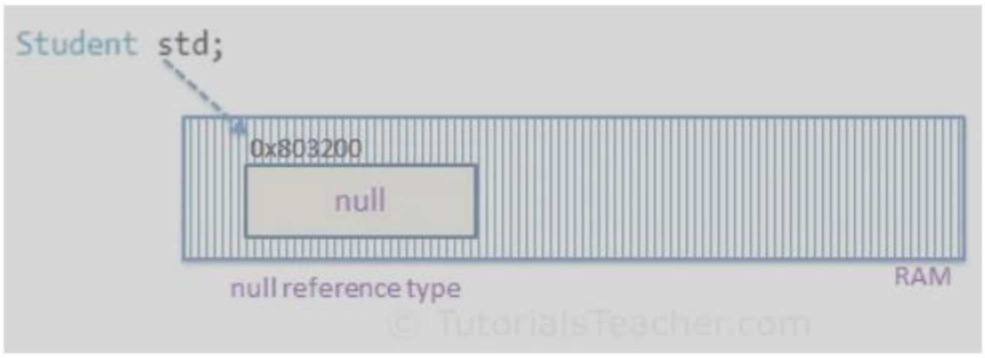
- String
- Class
- Delegates

Pass Reference Type by Value

- In c#, if we pass a reference type variable from one method to another method, the system won’t create a separate copy for that variable.
- Instead, it passes the address of the variable, so if we make any changes to the variable in one method, that also reflects in another method.

```
using System;
namespace CsharpExamples
{
  class Person
  {
    public int age;
  }
  class Program
  {
     static void Square(Person a, Person b)
     {
         a.age = a.age * a.age;
         b.age = b.age * b.age;
         Console.WriteLine(a.age + " " + b.age);
     }
     static void Main(string[] args)
     {
         Person p1 = new Person();
         Person p2 = new Person();
         p1.age = 5;
         p2.age = 10;
         Console.WriteLine(p1.age + " " + p2.age);
         Square(p1, p2);
         Console.WriteLine(p1.age + " " + p2.age);
         Console.WriteLine("Press Any Key to Exit..");
         Console.ReadLine();
     }
  }
}
```

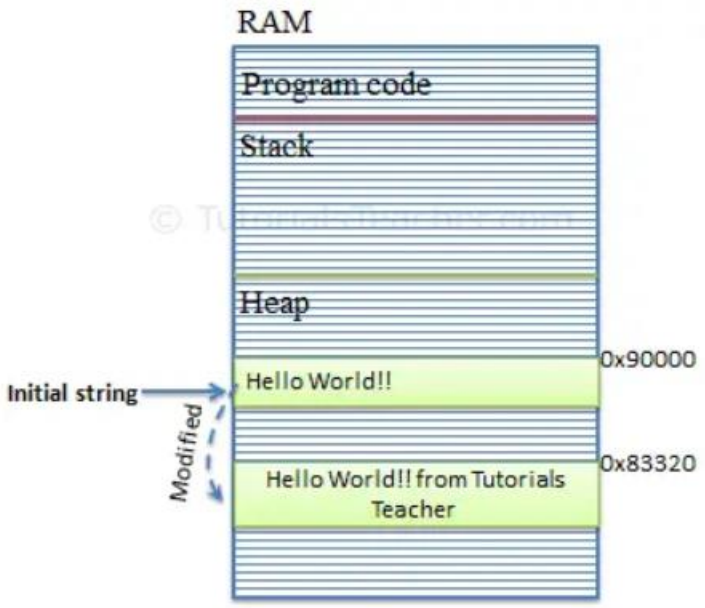
- The default value of a reference type variable is null when they are not initialized. Null means not refering to any object.



Null Reference Type

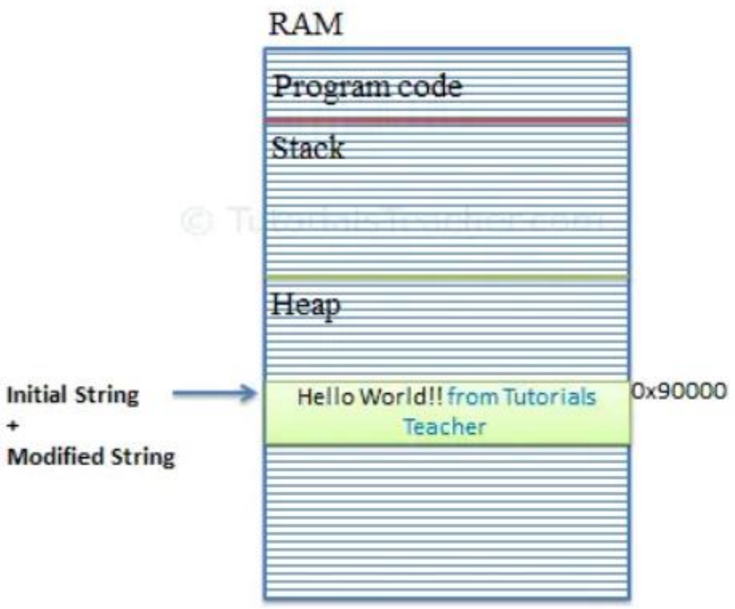
String v/s StringBuilder

- string type is immutable.
- It means a string cannot be changed once created. For example, a new string, "Hello World!" will occupy a memory space on the heap.
- Now, by changing the initial string "Hello World!" to "Hello World! from Tutorials Teacher" will create a new string object on the memory heap instead of modifying an original string at the same memory address.



Memory Allocation for String Object

- C# introduced the **StringBuilder** in the System.Text namespace.
- StringBuilder doesn't create a new object in the memory but dynamically expands memory to accommodate the modified string.



Memory Allocation for StringBuilder Object

Example: StringBuilder

```
using System.Text; // include at the top

StringBuilder sb = new StringBuilder(); //string will be appended later
//or
StringBuilder sb = new StringBuilder("Hello World!");
```

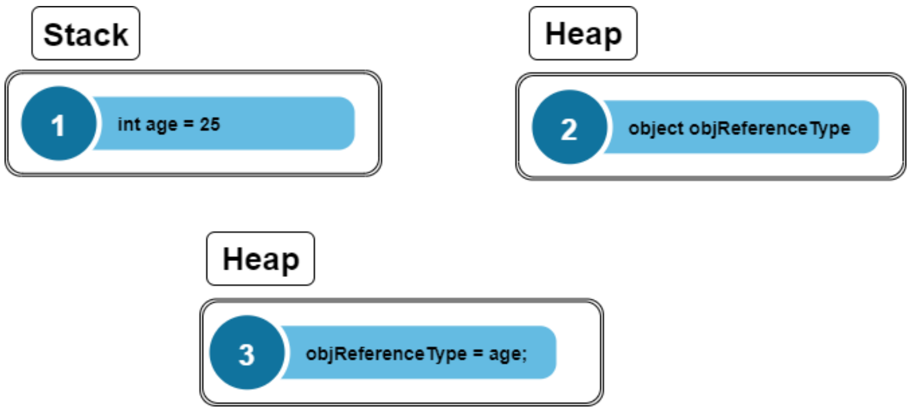
Nullable Types

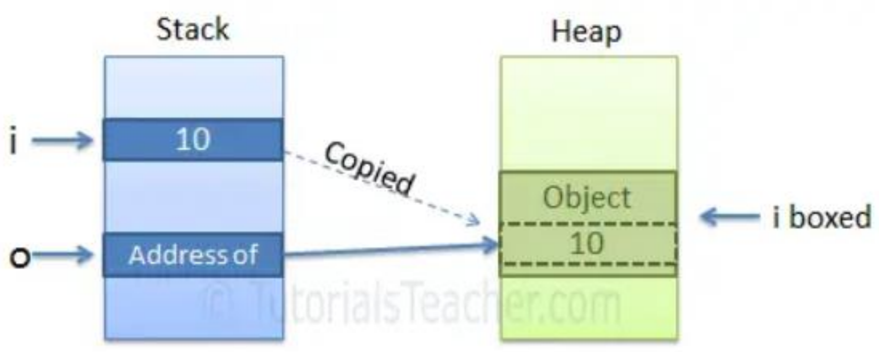
- Value type cannot be assigned a null value.
- example, int i = null will give you a compile time error.
- C# 2.0 introduced nullable types that allow you to assign null to value type variables. You can declare nullable types using Nullable<t> where T is a type.

Example: Nullable type
Nullable<int> i = null;

Boxing

- Boxing is the process of converting a value type to the object type or any interface type implemented by this value type. Boxing is **implicit**.
- Value type is always stored in a stack & referenced type is stored in a heap concept.
    1. int age = 25.
    2. object objReferenceType = age.
    3. objReferenceType = age.





Boxing

Example:

1) int i = 10;
   object o = i; //performs boxing

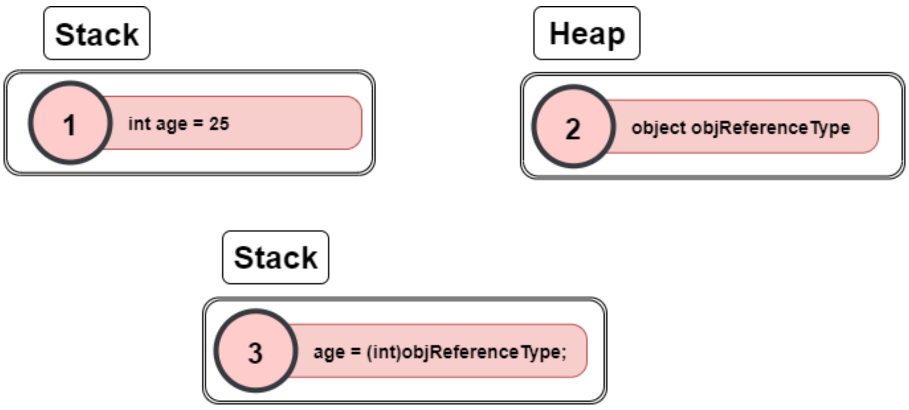
2) ArrayList list = new ArrayList();
   list.Add(10); // boxing
   list.Add("Bill");

- Step 1: declare n value type variable age, stored into a stack
- Step 2: declare an object type reference variable, stored into a heap
- Step 3: convertion of value type into reference type, which copies the value of age & stores into a reference type.

Unboxing:

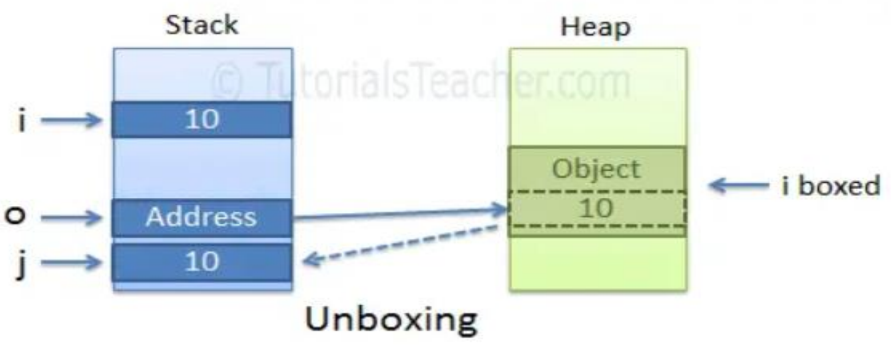
- Unboxing is the reverse of boxing. It is the process of converting a reference type to value type.
- Unboxing extract the value from the reference type and assign it to a value type.
- Unboxing is explicit. It means we must cast explicitly.

1. int age = 25.
2. object objReferenceType = new object ();
3. age = (int)objReferenceType.

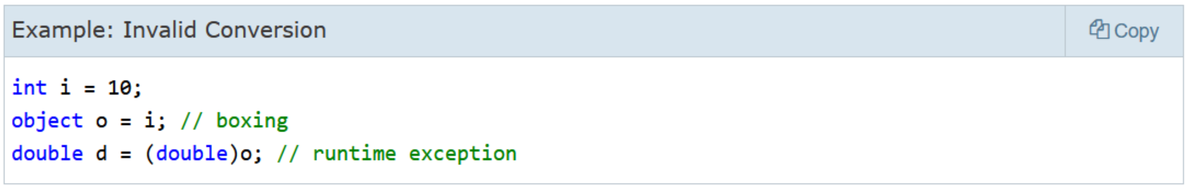


- Step 1: Declare a value type variable age, stored into stack
- Step 2: Declare an object type reference variable, stored into a heap
- Step 3: Conversion of reference type to value type by typecasting, which copies the reference of an object & stores into a value type.

```
object o = 10.7;
docuble i = (double)o; //performs unboxing
```

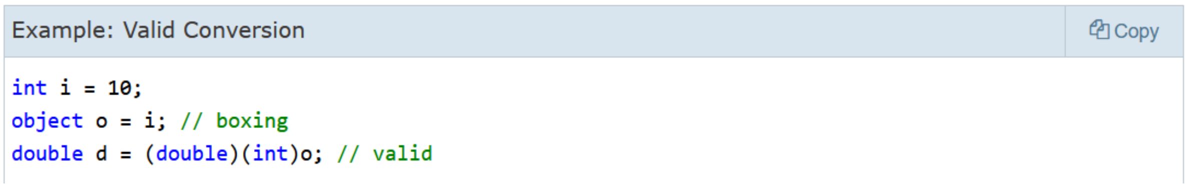


The casting of a boxed value is not permitted. The following will throw an exception.


Example: Invalid Conversion

```
int i = 10;
object o = i; // boxing
double d = (double)o; // runtime exception
```

First do unboxing and then do casting, as shown below.


Example: Valid Conversion

```
int i = 10;
object o = i; // boxing
double d = (double)(int)o; // valid
```

C# Type Casting

- Type casting is when you assign a value of one data type to another type.
- **Implicit Casting** (automatically) - converting a smaller type to a larger type size char -> int -> long -> float -> double.
- **Explicit Casting** (manually) - converting a larger type to a smaller size type double -> float -> long -> int -> char.

Example

1)
```
int myInt = 9;
double myDouble = myInt;       // Automatic casting: int to double
Console.WriteLine(myInt);      // Outputs 9
Console.WriteLine(myDouble);   // Outputs 9

double myDouble = 9.78;
int myInt = (int) myDouble;    // Manual casting: double to int
Console.WriteLine(myDouble);   // Outputs 9.78
Console.WriteLine(myInt);      // Outputs 9
```

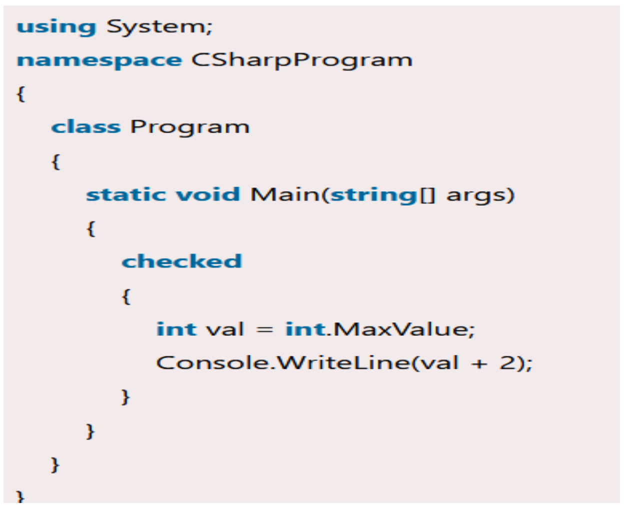
Type Conversion Methods

- It is also possible to convert data types explicitly by using built-in methods, such as **Convert.ToBoolean, convert.ToDouble, Convert.ToString, Convert.ToInt32 (int)** and **Convert.ToInt64** (long):

C# Checked and Unchecked

- C# provides checked and unchecked keyword to handle integral type exceptions.
- Checked and unchecked keywords specify checked context and unchecked context respectively.

C# Checked Example using checked

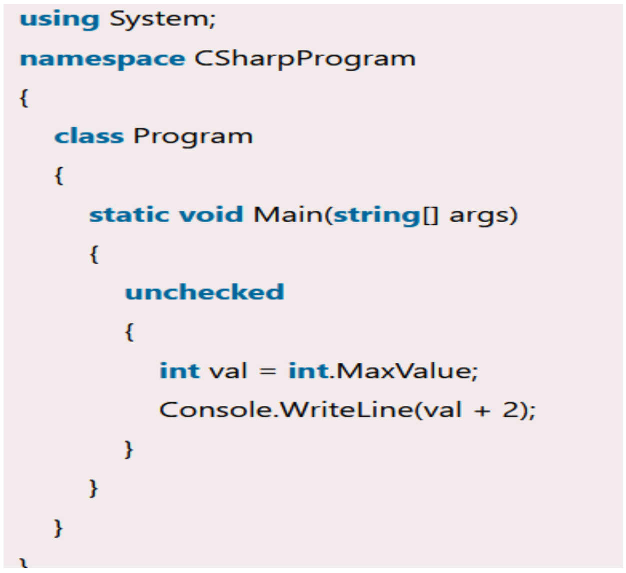


```
using System;
namespace CSharpProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            checked
            {
                int val = int.MaxValue;
                Console.WriteLine(val + 2);
            }
        }
    }
}
```

- Unhandled Exception: System.OverflowException: Arithmetic operation resulted in an overflow

C# Unchecked

- The Unchecked keyword ignores the integral type arithmetic exceptions.
- It does not check explicitly and produce result that may be truncated or wrong.



```
using System;
namespace CSharpProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            unchecked
            {
                int val = int.MaxValue;
                Console.WriteLine(val + 2);
            }
        }
    }
}
```

Output:

- -2147483647

C# Enumerations Type – Enum

- In C#, an enum (or enumeration type) is used to assign constant names to a group of numeric integers values.

```
enum WeekDays
{
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday,
   Sunday
}
```

Enum Values

- If values are not assigned to enum members, then the compiler will assign integer values to each member starting with zero by default.
- The first member of an enum will be 0, and the value of each successive enum member is increased by 1.

Example: Default Enum Values

```
enum WeekDays
{
   Monday,     // 0
   Tuesday,    // 1
   Wednesday,  // 2
   Thursday,   // 3
   Friday,     // 4
   Saturday,   // 5
   Sunday      // 6
}
```

Example: Assign Values to Enum Members

```
enum Categories
{
   Electronics,    // 0
   Food,           // 1
   Automotive = 6, // 6
   Arts,           // 7
   BeautyCare,     // 8
   Fashion         // 9
}
```

Example: Assign Values to Enum Members

```
enum Categories
{
   Electronics = 1,
   Food = 5,
   Automotive = 6,
   Arts = 10,
   BeautyCare = 11,
   Fashion = 15,
   WomanFashion = 15
}
```

Constants in C#

- Constants are immutable values which are known at compile time and do not change for the life of the program.
- Constants are declared with the const modifier.

```
class Converter
{
   public decimal KgToPound(decimal weight)
   {
      const decimal factor = 2.205m;
      return weight * factor;
   }
}
```

Control Statements in C#

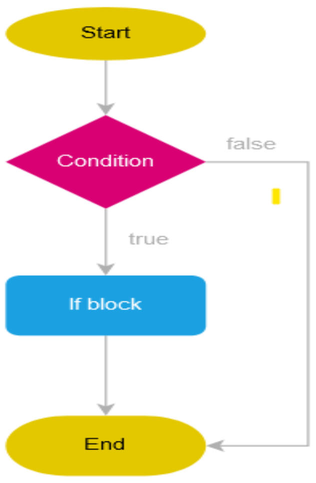
1) if statement

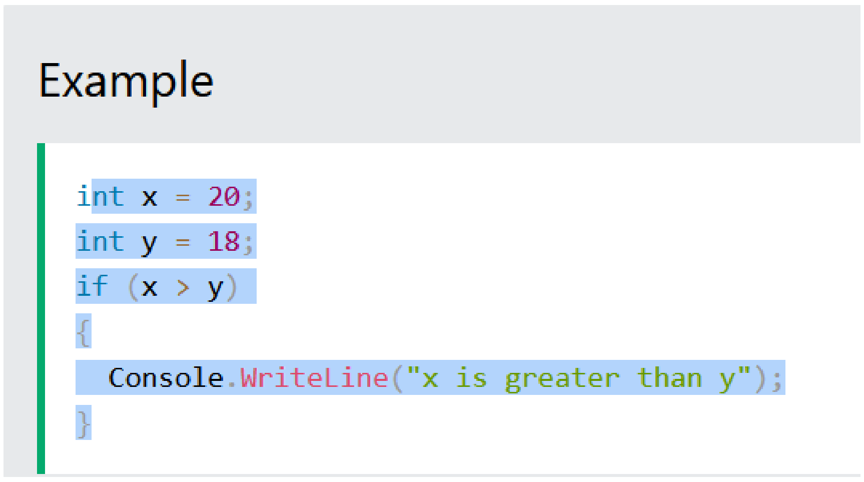
- The if statement evaluates a condition and executes one or more statements if the result is true.
- Otherwise, the if statement passes the control to the statement after it.

Example:

if (condition)

statement.



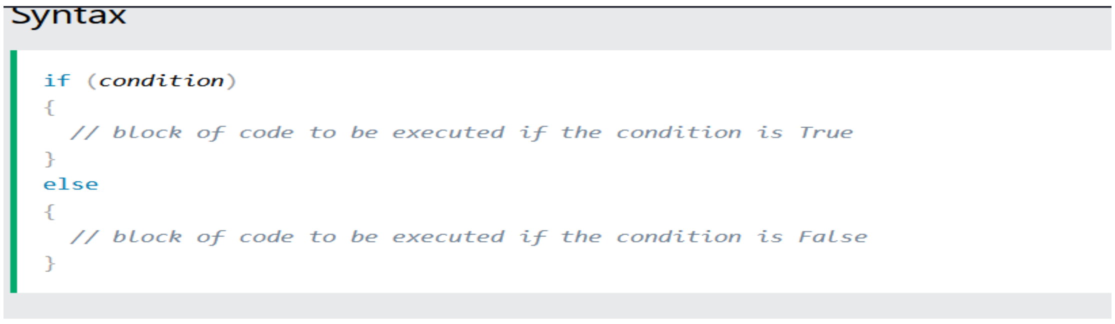


Example

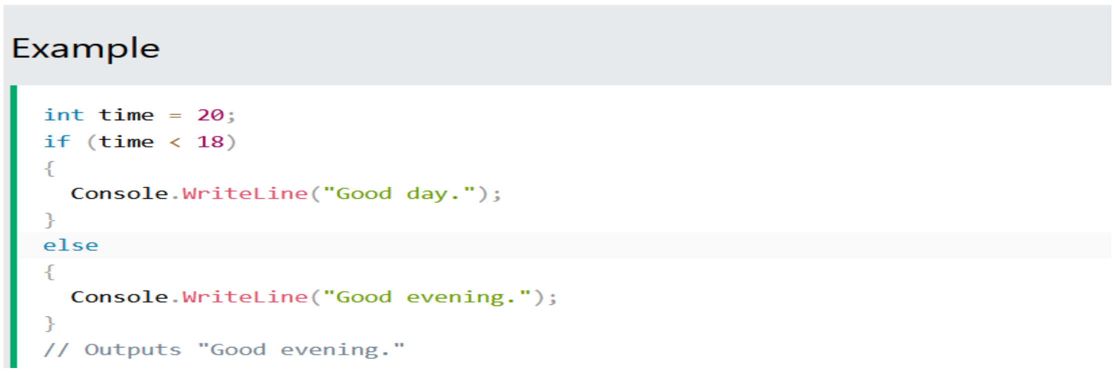
```
int x = 20;
int y = 18;
if (x > y)
{
  Console.WriteLine("x is greater than y");
}
```

2) The else Statement

- Specifies a block of code to be executed if the condition is False.


Syntax

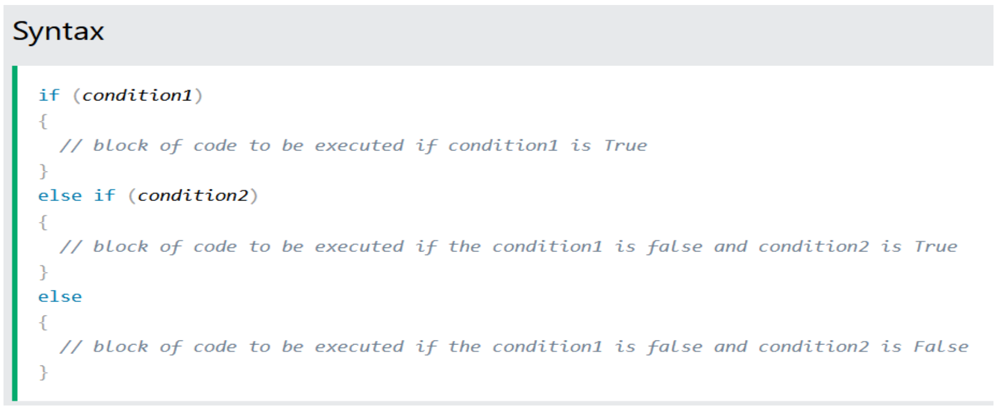
```
if (condition)
{
  // block of code to be executed if the condition is True
}
else
{
  // block of code to be executed if the condition is False
}
```


Example

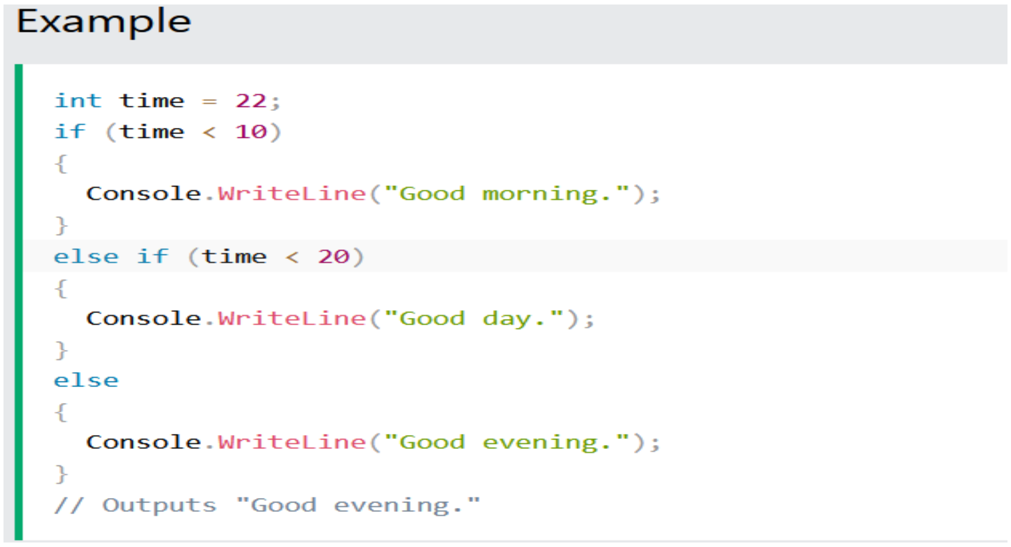
```
int time = 20;
if (time < 18)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
// Outputs "Good evening."
```

3) The else if Statement

- else if statement to specify a new condition if the first condition is False


Syntax

```
if (condition1)
{
  // block of code to be executed if condition1 is True
}
else if (condition2)
{
  // block of code to be executed if the condition1 is false and condition2 is True
}
else
{
  // block of code to be executed if the condition1 is false and condition2 is False
}
```


Example

```
int time = 22;
if (time < 10)
{
  Console.WriteLine("Good morning.");
}
else if (time < 20)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
// Outputs "Good evening."
```

4) Shorthand If...Else (Ternary Operator)

- There is also a shorthand if else, which is known as the **ternary operator** because it consists of three operands.
- It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

variable = (condition) ? expressionTrue : expressionFalse;

Example:

If Else

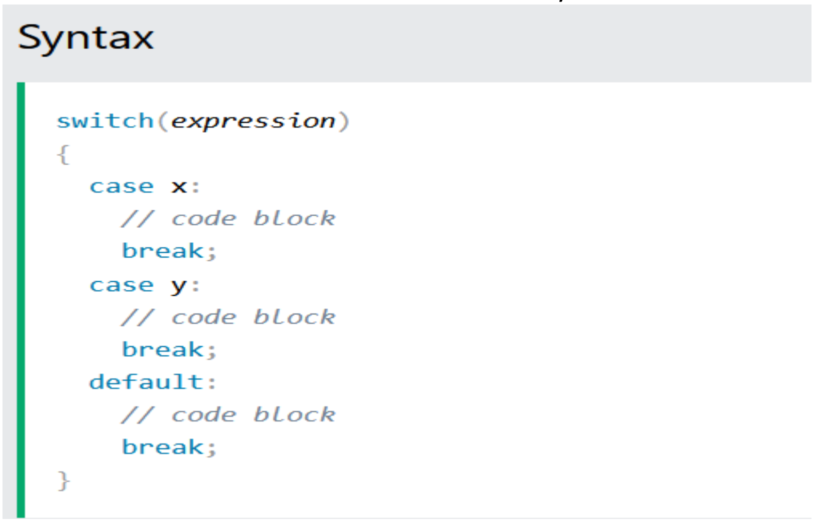
```
int time = 20;
if (time < 18)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
```

Ternary operator

```
int time = 20;
string result = (time < 18) ? "Good day." : "Good evening.";
Console.WriteLine(result);
```

5) Switch Statements

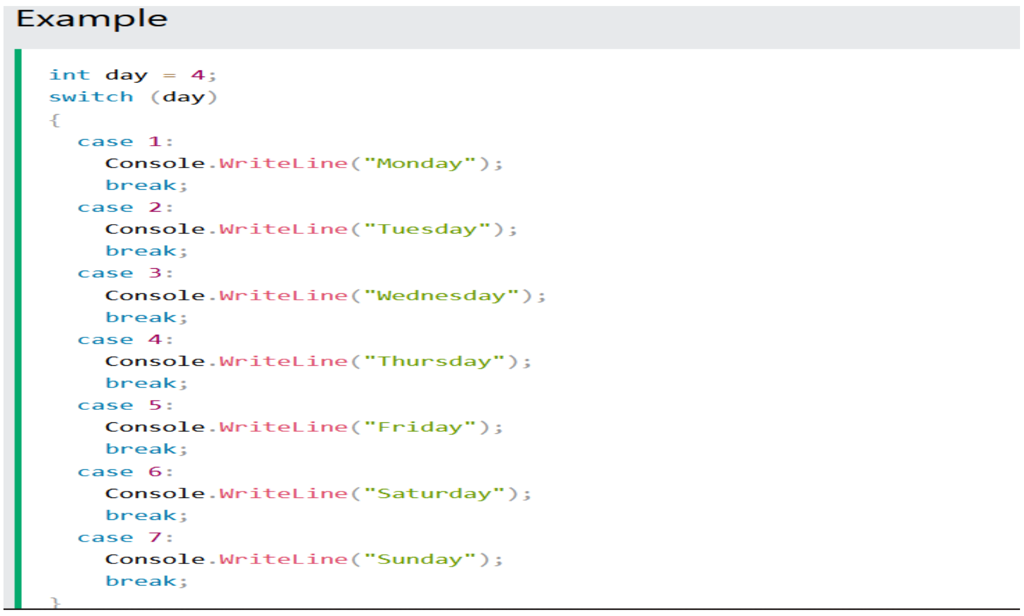
- Use the switch statement to select one of many code blocks to be executed.



Syntax

```
switch(expression)
{
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
    break;
}
```

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.



Example

```
int day = 4;
switch (day)
{
  case 1:
    Console.WriteLine("Monday");
    break;
  case 2:
    Console.WriteLine("Tuesday");
    break;
  case 3:
    Console.WriteLine("Wednesday");
    break;
  case 4:
    Console.WriteLine("Thursday");
    break;
  case 5:
    Console.WriteLine("Friday");
    break;
  case 6:
    Console.WriteLine("Saturday");
    break;
  case 7:
    Console.WriteLine("Sunday");
    break;
}
```

Loops

- Loops can execute a block of code if a specified condition is reached.
- Loops are handy because they save time, reduce errors, and they make code more readable.

1) While Loop

- The while loop loops through a block of code if a specified condition is True:

Syntax

```
while (condition)
{
// code block to be executed
}
```

Example

```
int i = 0;
while (i < 5)
{
Console.WriteLine(i);
i++;
}
```

2) Do/While Loop

- **Do/while** loop is a variant of the while loop.
- loop will execute the code block once, before checking if the condition is true, then it will repeat the loop if the condition is true.

Syntax

```
do
{
// code block to be executed
}
while (condition);
```

Example

```
int i = 0;
do
{
Console.WriteLine(i);
i++;
}
while (i < 5);
```

C# For Loop

- When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop

Syntax

```
for (statement 1; statement 2; statement 3)
{
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

Example:

```
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

Nested Loops

- It is also possible to place a loop inside another loop. This is called a **nested loop**.
- The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

```
// Outer loop
for (int i = 1; i <= 2; ++i)
{
  Console.WriteLine("Outer: " + i);  // Executes 2 times

  // Inner loop
  for (int j = 1; j <= 3; j++)
  {
    Console.WriteLine(" Inner: " + j); // Executes 6 times (2 * 3)
  }
}
```

Foreach Loop

- foreach loop, which is used exclusively to loop through elements in an **array.**

Syntax

```
foreach (type variableName in arrayName)
{
  // code block to be executed
}
```

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
foreach (string i in cars)
{
  Console.WriteLine(i);
}
```